IT-314 Software Engineering

Lab - 07 Program Inspection, Debugging and Static Analysis

Name : Shyam Ghetiya

Student-Id : 202201161

Aim:

Prerequisites: Before the lab you should check that the Eclipse/NetBeans IDE is being installed in the lab system.

Preparation: Review ten code fragments given as the text files using two ways, (1) the code is reviewed using the inspection (checklist-based approach) technique (2) the debugger in the IDE is used to identify the error and review the code fragment

Activities: Having considered the code fragments you should carry out the following activities that will be facilitated by your TA/Course Instructor.

Goal : To answer the following questions - Program Inspection: (Submit the answers of following questions for each code fragment)

Program inspection :

1. How many errors are there in the program? Mention the errors you have identified.
2. Which category of program inspection would you find more effective?
3. Which type of error you are not able to identified using the program inspection?
4. Is the program inspection technique is worth applicable?

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

2. How many breakpoints do you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

3. Submit your complete executable code?

Program Inspection, Debugging

Answer :

1. GCD And LCM :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- Error in the GCD calculation logic:
  - The while loop in the gcd function incorrectly uses == instead of !=. This causes the loop to execute incorrectly for many inputs.
  - Fix: Change while(a % b == 0) to while(a % b != 0).
- Error in the LCM calculation logic:
  - The condition in the lcm function is wrong. It should check when a is divisible by both x and y, not when it's not divisible.

    - Fix: Change if(a % x != 0 && a % y != 0) to if(a % x == 0 && a % y == 0).

2. Which category of program inspection would you find more effective?

- Control Flow and Logical Errors: By focusing on the control flow and logical correctness, both errors in the gcd and lcm methods were identified. The gcd loop control and the condition in lcm were both problematic.

3. Which type of error were you not able to identify using program inspection?

- Run-time Errors and Edge Case Issues: Program inspection cannot directly identify run-time issues or how the code behaves for all edge cases without running the program (like extremely large values or specific edge cases like GCD(0, n)).

4. Is the program inspection technique worth applying?

- Yes, program inspection helps to catch obvious logical and control flow errors early in the process, reducing the number of issues that need to be addressed during debugging.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Two major errors were found:
  - Error 1: Incorrect condition in the gcd method (while(a % b == 0)).
  - Error 2: Incorrect logic in the lcm method (if(a % x != 0 && a % y != 0)).

2. How many breakpoints did you need to fix those errors?

- Two breakpoints:
  - One at the start of the while loop in the gcd function to inspect how the loop runs.
  - Another inside the while(true) loop in the lcm function to check the condition when the least common multiple is found.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: For the gcd method, I changed the while loop condition to while(a % b != 0) to ensure it loops until the remainder is 0.
- Fix 2: In the lcm method, I corrected the condition to if(a % x == 0 && a % y == 0) to correctly find the least common multiple.

3. Submit your complete executable code:

```
import java.util.Scanner;

public class GCD_LCM
```

```
{
   static int gcd(int x, int y)
   {
       int r = 0, a, b;
       a = (x > y) ? x : y; // a is the greater number
       b = (x < y) ? x : y; // b is the smaller number
       r = b;
       while(a % b != 0) // Corrected: != instead of ==
       {
           r = a % b;
           a = b;
           b = r;
       }
       return r;
   }
   static int lcm(int x, int y)
   {
       int a;
       a = (x > y) ? x : y; // a is the greater number
       while(true)
       {
           if(a % x == 0 && a % y == 0) // Corrected: == instead of !=
               return a;
           ++a;
```

```java
        }

    }

    public static void main(String args[])

    {

        Scanner input = new Scanner(System.in);

        System.out.println("Enter the two numbers: ");

        int x = input.nextInt();

        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " + gcd(x, y));

        System.out.println("The LCM of two numbers is: " + lcm(x, y));

        input.close();

    }

}
```

Input :

Enter the two numbers: 4 5

Output :

The GCD of two numbers is: 1

The LCM of two numbers is: 20

2 . Sorting Array :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- Error in the outer loop condition:
  - The condition in the outer loop for (int i = 0; i >= n; i++); is incorrect. It should use i < n instead of i >= n to ensure it iterates over all elements.
  - Fix: Change for (int i = 0; i >= n; i++); to for (int i = 0; i < n; i++).
- Error in the sorting logic:
  - The sorting logic is incorrect; it should be designed to arrange the elements in ascending order. The current logic swaps the elements incorrectly.
  - The sorting algorithm should compare a[i] > a[j] to swap them if the first is greater than the second.
  - Fix: Change the condition in the if statement to if (a[i] > a[j]).

2. Which category of program inspection would you find more effective?

- Control Flow and Logical Errors: By focusing on the control flow and logical correctness, the issues in the loop conditions and sorting logic were identified.

3. Which type of error were you not able to identify using program inspection?

- Edge Case Issues: Program inspection cannot directly identify how the code behaves with specific edge cases, such as when the input array is already sorted or contains duplicate values.

4. Is the program inspection technique worth applying?

- Yes, program inspection helps catch obvious logical errors and control flow issues early, reducing the time spent on debugging.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Two major errors were found:
  - Error 1: Incorrect condition in the outer loop (i >= n).
  - Error 2: Incorrect sorting condition (a[i] <= a[j]).

2. How many breakpoints did you need to fix those errors?

- Two breakpoints:

- One at the start of the outer loop to check the loop's execution.
- Another inside the inner loop to verify the swapping condition and logic.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: For the outer loop, change the condition to i < n to ensure it iterates correctly.
- Fix 2: Updated the condition in the inner loop to if (a[i] > a[j]) to correctly implement ascending order sorting.

3. Submit your complete executable code:

```
import java.util.Scanner;

public class Ascending_Order

{

   public static void main(String[] args)

   {

       int n, temp;

       Scanner s = new Scanner(System.in);

       System.out.print("Enter no. of elements you want in array: ");

       n = s.nextInt();

       int a[] = new int[n];

       System.out.println("Enter all the elements:");

       for (int i = 0; i < n; i++)
```

```java
        {
            a[i] = s.nextInt();
        }
        for (int i = 0; i < n; i++) // Corrected: i < n instead of i >= n
        {
            for (int j = i + 1; j < n; j++)
            {
                if (a[i] > a[j]) // Corrected: > instead of <=
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++)
        {
            System.out.print(a[i] + ",");
        }
        System.out.print(a[n - 1]);
    }
}
```

Input :

Enter no. of elements you want in array: 5

Enter all the elements: 1 12 2 9 7

Output :

Ascending Order: 1,2,7,9,12

3 . Merge sort :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- **Error in array slicing:**
  - **The methods leftHalf(array + 1) and rightHalf(array - 1) are incorrect because you cannot perform arithmetic operations directly on arrays. Instead, the methods should accept the actual array.**
  - **Fix: Change leftHalf(array + 1) to leftHalf(array) and rightHalf(array - 1) to rightHalf(array).**
- **Error in merge function call:**
  - **The call merge(array, left++, right--); is incorrect. The ++ and -- operators are not valid in this context, and they should not be used.**

    - Fix: Change to merge(array, left, right);.
- Error in the leftHalf method:
    - The calculation of the size for the left half does not handle odd-length arrays correctly.
    - Fix: Update int size1 = array.length / 2; to int size1 = (array.length + 1) / 2; to account for odd lengths.

2. Which category of program inspection would you find more effective?

- Control Flow and Logical Errors: These types of errors are crucial in recursive algorithms like merge sort, where the flow of execution and how the data is processed are key.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: Issues such as null pointer exceptions or index out-of-bounds errors may not be identified during static inspection but would appear at runtime.

4. Is the program inspection technique worth applying?

- Yes, program inspection is beneficial for identifying logical and syntax errors before runtime, making debugging more efficient.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three major errors were found:
  - Error 1: Incorrect array slicing in leftHalf and rightHalf.
  - Error 2: Incorrect usage of ++ and -- in the merge method call.
  - Error 3: Incorrect handling of odd-length arrays in leftHalf.

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:
  - One at the beginning of the mergeSort method to check how the array is being split.
  - One inside the merge method to check the merging logic.
  - One inside the leftHalf method to verify how the array is divided.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Modified leftHalf and rightHalf calls to pass the actual array.
- Fix 2: Removed the incorrect ++ and -- operators in the merge function call.

- **Fix 3: Adjusted the left half size calculation to handle odd lengths correctly.**

3. Submit your complete executable code:

```
// This program implements the merge sort algorithm for

// arrays of integers.

import java.util.*;

public class MergeSort {

   public static void main(String[] args) {

       int[] list = {14, 32, 67, 76, 23, 41, 58, 85};

       System.out.println("before: " + Arrays.toString(list));

       mergeSort(list);

       System.out.println("after:  " + Arrays.toString(list));

   }

   // Places the elements of the given array into sorted order

   // using the merge sort algorithm.

   // post: array is in sorted (nondecreasing) order

   public static void mergeSort(int[] array) {

       if (array.length > 1) {

           // split array into two halves

           int[] left = leftHalf(array);

           int[] right = rightHalf(array);

           // recursively sort the two halves

           mergeSort(left);
```

```java
        mergeSort(right);

        // merge the sorted halves into a sorted whole

        merge(array, left, right);

    }

}

// Returns the first half of the given array.

public static int[] leftHalf(int[] array) {

    int size1 = (array.length + 1) / 2; // Corrected to handle odd lengths

    int[] left = new int[size1];

    for (int i = 0; i < size1; i++) {

        left[i] = array[i];

    }

    return left;

}

// Returns the second half of the given array.

public static int[] rightHalf(int[] array) {

    int size1 = array.length / 2;

    int size2 = array.length - size1;

    int[] right = new int[size2];

    for (int i = 0; i < size2; i++) {

        right[i] = array[i + size1];

    }

    return right;

}
```

```java
    // Merges the given left and right arrays into the given
   // result array.  Second, working version.
   // pre : result is empty; left/right are sorted
   // post: result contains result of merging sorted lists;
   public static void merge(int[] result,
                            int[] left, int[] right) {
       int i1 = 0;   // index into left array
       int i2 = 0;   // index into right array
       for (int i = 0; i < result.length; i++) {
           if (i2 >= right.length || (i1 < left.length &&
                   left[i1] <= right[i2])) {
               result[i] = left[i1];    // take from left
               i1++;
           } else {
               result[i] = right[i2];   // take from right
               i2++;
           }
       }
   }
}
```

Input :

before: [14, 32, 67, 76, 23, 41, 58, 85]

Output :

after: [14, 23, 32, 41, 58, 67, 76, 85]

4 . Magic number :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- Control Flow Error:
  - The condition while(sum == 0) in the inner while loop is incorrect. It should be while(sum != 0) to continue processing as long as there are digits to sum.
  - Fix: Change while(sum == 0) to while(sum != 0).
- Logical Error in Sum Calculation:
  - The calculation s = s * (sum / 10) is incorrect. It should be s += sum % 10 to correctly sum the digits of sum.
  - Fix: Change s = s * (sum / 10) to s += sum % 10.
- Syntax Error:
  - The statement sum=sum%10 is missing a semicolon at the end.
  - Fix: Add a semicolon after sum = sum % 10.

2. Which category of program inspection would you find more effective?

- Control Flow and Logical Errors: These types of errors are essential in determining whether the logic for checking if a number is a magic number is implemented correctly.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: Certain runtime errors such as division by zero or infinite loops may not be caught during static inspection.

4. Is the program inspection technique worth applying?

- Yes, program inspection is valuable for identifying logical and syntax errors before runtime, helping streamline the debugging process.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three major errors were found:
  - Error 1: Incorrect condition in the inner while loop (while(sum == 0)).
  - Error 2: Incorrect summation logic (s = s * (sum / 10)).
  - Error 3: Missing semicolon (sum = sum % 10).

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:
  - One at the beginning of the while loop that processes the digits.
  - One inside the inner while loop to check the summation logic.

- One at the output statement to confirm the final result.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Updated the while loop condition to while(sum != 0).
- Fix 2: Corrected the summation logic to s += sum % 10.
- Fix 3: Added a semicolon after sum = sum % 10.

3. Submit your complete executable code :

```
// Program to check if number is Magic number in JAVA

import java.util.*;

public class MagicNumberCheck {

   public static void main(String args[]) {

       Scanner ob = new Scanner(System.in);

       System.out.println("Enter the number to be checked.");

       int n = ob.nextInt();

       int sum = 0, num = n;

       while (num > 9) {

           sum = num;

           int s = 0;

           while (sum != 0) { // Fixed condition

               s += sum % 10; // Fixed summation logic

               sum = sum / 10; // Added semicolon

           }
```

```
            num = s;

        }

        if (num == 1) {

            System.out.println(n + " is a Magic Number.");

        } else {

            System.out.println(n + " is not a Magic Number.");

        }

    }

}
```

Input :

Enter the number to be checked: 119

Output :

119 is a Magic Number.

<u>5. Multiply matrix :</u>

Program Inspection

1. How many errors are there in the program? Mention the errors you have identified.

- **Logical Errors:**

- Error 1: The loop indices for accessing the first and second matrices are incorrect. The index should be first[c][k] and second[k][d], not using -1.
  - Fix: Change first[c-1][c-k] to first[c][k] and second[k-1][k-d] to second[k][d].
- Error 2: The program incorrectly prompts for the number of rows and columns of the second matrix twice.
  - Fix: The second prompt should be updated to say "Enter the number of rows and columns of second matrix."
- Error 3: The output formatting does not maintain alignment and readability for matrix display.
  - Fix: Update the print statement to use printf for better formatting.

2. Which category of program inspection would you find more effective?

- Logical and Syntax Errors: These errors affect the program's execution and correctness, making it essential to verify the logic for matrix multiplication.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: Issues such as array index out of bounds or incorrect input sizes may only manifest during execution, making them difficult to spot in static inspection.

4. Is the program inspection technique worth applying?

- Yes, program inspection helps in identifying logical and syntax errors early, reducing the likelihood of runtime failures.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three major errors were identified:
  - Error 1: Incorrect indexing in matrix multiplication.
  - Error 2: Incorrect prompt for second matrix dimensions.
  - Error 3: Output formatting issue.

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:
  - One before the multiplication logic to check the matrix values.
  - One at the dimension prompts to ensure correct user input.
  - One before printing the result to check the product matrix.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Updated the indexing in the multiplication loop.
- Fix 2: Corrected the prompt for the second matrix dimensions.
- Fix 3: Used formatted output for better readability.

3. Submit your complete executable code:

```
// Java program to multiply two matrices

import java.util.Scanner;

class MatrixMultiplication {

   public static void main(String args[]) {

       int m, n, p, q, sum, c, d, k;

       Scanner in = new Scanner(System.in);

       System.out.println("Enter the number of rows and columns of first matrix");

       m = in.nextInt();

       n = in.nextInt();

       int first[][] = new int[m][n];

       System.out.println("Enter the elements of first matrix");

       for (c = 0; c < m; c++)

           for (d = 0; d < n; d++)

               first[c][d] = in.nextInt();

       System.out.println("Enter the number of rows and columns of second matrix");

       p = in.nextInt();

       q = in.nextInt();
```

```
        if (n != p) {
            System.out.println("Matrices with entered orders can't be multiplied with
each other.");
        } else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];
            System.out.println("Enter the elements of second matrix");
            for (c = 0; c < p; c++)
                for (d = 0; d < q; d++)
                    second[c][d] = in.nextInt();
            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    sum = 0; // Initialize sum for each cell
                    for (k = 0; k < n; k++) {
                        sum += first[c][k] * second[k][d]; // Corrected indexing
                    }
                    multiply[c][d] = sum;
                }
            }
            System.out.println("Product of entered matrices:");
            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    System.out.print(multiply[c][d] + "\t"); // Improved formatting
```

```
                }

                System.out.print("\n");

            }

        }

    }

}
```

Input :

Enter the number of rows and columns of first matrix

2 2

Enter the elements of first matrix

1 2

3 4

Enter the number of rows and columns of second matrix

2 2

Enter the elements of second matrix

1 0

1 0

Output :

Product of entered matrices:

3 0

7 0

6. Quadratic Probing :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- Logical Errors:
  - Error 1: In the insert method, the line i + = (i + h / h--) % maxSize; has incorrect syntax with a space between + and =. It should be i += (i + h) % maxSize;.
  - Error 2: The rehashing loop in the remove method has similar issues with indexing, as it incorrectly applies h * h++ instead of the correct quadratic probing formula.
  - Error 3: The variable maxSizeake is incorrectly named; it should be maxSize.

2. Which category of program inspection would you find more effective?

- Logical Errors: Since they can lead to incorrect functioning, logical errors must be thoroughly inspected and tested to ensure the algorithm works as expected.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: These can occur due to improper handling of edge cases, such as inserting when the hash table is full or using an invalid key.

4. Is the program inspection technique worth applying?

- Yes, program inspection is crucial for identifying logical and syntax errors early, preventing runtime issues during execution.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three major errors were identified:
  - Error 1: Incorrect syntax in the insert method.
  - Error 2: Issues in the rehashing logic during the remove method.
  - Error 3: Incorrect naming in the comments.

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:
  - One in the insert method to inspect the key-value insertion logic.
  - One in the remove method to check the rehashing logic.
  - One in the get method to ensure correct retrieval.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Corrected the syntax in the insert method.
- Fix 2: Updated the rehashing logic to apply correct quadratic probing.

- **Fix 3: Revised comments and naming for clarity.**

3. Submit your complete executable code:

```
/**
*   Java Program to implement Quadratic Probing Hash Table
**/
import java.util.Scanner;
/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
   private int currentSize, maxSize;
   private String[] keys;
   private String[] vals;
   /** Constructor **/
   public QuadraticProbingHashTable(int capacity) {
       currentSize = 0;
       maxSize = capacity;
       keys = new String[maxSize];
       vals = new String[maxSize];
   }
   /** Function to clear hash table **/
   public void makeEmpty() {
       currentSize = 0;
       keys = new String[maxSize];
       vals = new String[maxSize];
```

```java
}

/** Function to get size of hash table **/

public int getSize() {

    return currentSize;

}

/** Function to check if hash table is full **/

public boolean isFull() {

    return currentSize == maxSize;

}

/** Function to check if hash table is empty **/

public boolean isEmpty() {

    return getSize() == 0;

}

/** Function to check if hash table contains a key **/

public boolean contains(String key) {

    return get(key) != null;

}

/** Function to get hash code of a given key **/

private int hash(String key) {

    return Math.abs(key.hashCode()) % maxSize;

}

/** Function to insert key-value pair **/

public void insert(String key, String val) {

    int tmp = hash(key);
```

```
    int i = tmp, h = 1;

    do {

        if (keys[i] == null) {

            keys[i] = key;

            vals[i] = val;

            currentSize++;

            return;

        }

        if (keys[i].equals(key)) {

            vals[i] = val;

            return;

        }

        i += (h * h) % maxSize; // Corrected indexing logic

        h++; // Increment h for quadratic probing

    } while (i != tmp);

}

/** Function to get value for a given key **/

public String get(String key) {

    int i = hash(key), h = 1;

    while (keys[i] != null) {

        if (keys[i].equals(key))

            return vals[i];

        i = (i + h * h) % maxSize; // Corrected indexing logic

        h++; // Increment h for quadratic probing
```

```
    }
    return null;
}
/** Function to remove key and its value **/
public void remove(String key) {
    if (!contains(key))
        return;
    /** find position key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h) % maxSize;
    keys[i] = vals[i] = null;
    /** rehash all keys **/
    for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h) % maxSize) {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
    currentSize--;
}


/** Function to print HashTable **/
public void printHashTable() {
```

```
        System.out.println("\nHash Table: ");

        for (int i = 0; i < maxSize; i++)

            if (keys[i] != null)

                System.out.println(keys[i] + " " + vals[i]);

        System.out.println();

    }

}

/** Class QuadraticProbingHashTableTest **/

public class QuadraticProbingHashTableTest {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);

        System.out.println("Hash Table Test\n\n");

        System.out.println("Enter size");

        /** Create object of QuadraticProbingHashTable **/

        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

        char ch;

        /**  Perform QuadraticProbingHashTable operations  **/

        do {

            System.out.println("\nHash Table Operations\n");

            System.out.println("1. insert ");

            System.out.println("2. remove");

            System.out.println("3. get");

            System.out.println("4. clear");

            System.out.println("5. size");
```

```
        int choice = scan.nextInt();

        switch (choice) {

            case 1 :

                System.out.println("Enter key and value");

                qpht.insert(scan.next(), scan.next());

                break;

            case 2 :

                System.out.println("Enter key");

                qpht.remove(scan.next());

                break;

            case 3 :

                System.out.println("Enter key");

                System.out.println("Value = " + qpht.get(scan.next()));

                break;

            case 4 :

                qpht.makeEmpty();

                System.out.println("Hash Table Cleared\n");

                break;

            case 5 :

                System.out.println("Size = " + qpht.getSize());

                break;

            default :

                System.out.println("Wrong Entry \n ");

                break;
```

```
            }

            /** Display hash table **/

            qpht.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n) \n");

            ch = scan.next().charAt(0);

        } while (ch == 'Y' || ch == 'y');

    }

}
```

Input :

Hash Table Test

Enter size: 5

Hash Table Operations

1. insert

2. remove

3. get

4. clear

5. size

1

Enter key and value

c computer

d desktop

h harddrive

Output :

Hash Table:
c computer
d desktop
h harddrive

7. Stack implementation :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- Logical Errors:
  - Error 1: In the push method, the line top--; should be top++; to increment the top index before storing the value. The current implementation is incorrect as it decreases the top index, which causes it to point to an invalid position.
  - Error 2: The loop condition in the display method is incorrect: it should be i <= top instead of i > top. This means the loop will not execute correctly.
  - Error 3: The stack implementation should not use negative indexing; hence, initializing top to -1 is correct, but the incrementing should be done appropriately during the push.

2. Which category of program inspection would you find more effective?

- Logical Errors: These can lead to incorrect behavior during execution, so careful examination of the stack operations (push and pop) is crucial.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: These may arise from trying to push or pop from an empty or full stack.

4. Is the program inspection technique worth applying?

- Yes, it is essential for identifying logical errors and ensuring that the stack operations are functioning correctly.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three errors were identified:
    - Incorrect indexing in the push method.
    - Incorrect loop condition in the display method.
    - Logic in the pop method to handle an empty stack.

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:

- One in the push method to inspect the value being pushed.
- One in the pop method to check the behavior when the stack is empty.
- One in the display method to ensure all elements are printed.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Changed top--; to top++; in the push method.
- Fix 2: Updated the loop condition in the display method to i <= top.
- Fix 3: Adjusted the logic in the pop method to decrement the top index appropriately.

3. Submit your complete executable code:

```
// Stack implementation in java
import java.util.Arrays;

public class StackMethods {
   private int top;
   int size;
   int[] stack;

   public StackMethods(int arraySize) {
       size = arraySize;
       stack = new int[size];
       top = -1;
   }

   public void push(int value) {
       if (top == size - 1) {
           System.out.println("Stack is full, can't push a value");
       } else {
           top++; // Increment top before adding the value
           stack[top] = value; // Store the value at the top index
```

```
        }
    }

    public void pop() {
        if (!isEmpty()) {
            System.out.println("Popped value: " + stack[top]); // Optional: Show popped
value
            top--; // Decrement top to remove the value
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
        } else {
            System.out.print("Stack elements: ");
            for (int i = 0; i <= top; i++) { // Changed the condition to <=
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display();
```

```
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();
    }
}
```

Output :
Stack elements: 10 1 50 20 90
Popped value: 90
Popped value: 20
Popped value: 50
Popped value: 1
Stack is empty.

8. Knapsack :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.

- **Logical Errors:**
  - **Error 1: The increment of n in the line int option1 = opt[n++][w]; causes n to exceed the number of items, leading to an ArrayIndexOutOfBoundsException.**

- Error 2: In the calculation of option2, the index profit[n-2] should be profit[n], which may lead to incorrect profit calculation for the last item.
- Error 3: The weight check if (weight[n] > w) should be corrected to allow the condition to consider whether the current item can be included in the knapsack.

2. Which category of program inspection would you find more effective?

- Logical Errors: These errors will prevent the program from functioning as expected and need to be addressed for correct execution.

3. Which type of error were you not able to identify using program inspection?

- Runtime Errors: These may arise from invalid array accesses due to the incorrect handling of indices.

4. Is the program inspection technique worth applying?

- Yes, it is essential for identifying logical errors and ensuring that the algorithm works correctly.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.

- Three errors were identified:
    - Incorrect index handling in the dynamic programming table.
    - Incorrect reference to the profit array in the option2 calculation.
    - Logic error in the weight comparison condition.

2. How many breakpoints did you need to fix those errors?

- Three breakpoints:
    - One in the option calculation to inspect the current state of n and w.
    - One in the profit calculation to check the values being referenced.
    - One in the weight condition check to verify item inclusion.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Changed int option1 = opt[n++][w]; to int option1 = opt[n][w]; to avoid incrementing n.
- Fix 2: Updated int option2 = profit[n-2] + opt[n-1][w-weight[n]]; to int option2 = profit[n] + opt[n-1][w-weight[n]];.
- Fix 3: Changed the condition in if (weight[n] > w) to if (weight[n] <= w).

3. Submit your complete executable code:

```
// Knapsack
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);    // number of items
        int W = Integer.parseInt(args[1]);    // maximum weight of knapsack

        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight limit w
        // sol[n][w] = does opt solution to pack items 1..n with weight limit w include
item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {

                // don't take item n
                int option1 = opt[n][w]; // Corrected increment

                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) { // Updated condition
                    option2 = profit[n] + opt[n-1][w-weight[n]]; // Corrected index
                }

                // select better of two options
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
```

```
        }

        // determine which items to take
        boolean[] take = new boolean[N+1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w = w - weight[n];
            } else {
                take[n] = false;
            }
        }

        // print results
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" +
take[n]);
        }
    }
}
```

Output :

item	profit	weight	take
1	336	784	false
2	674	1583	false
3	763	392	true
4	544	1136	true
5	14	1258	false
6	738	306	true

<u>9 . Tower of Hanoi :</u>

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.
   - Logical Errors:
     - Error 1: The use of topN++ and inter-- in the recursive call doTowers(topN++, inter--, from+1, to+1) is incorrect because it modifies the values instead of passing the current values for the recursive calls.
     - Error 2: The from + 1 and to + 1 expressions should not increment the from and to parameters as they represent the names of the pegs, which should remain unchanged.
2. Which category of program inspection would you find more effective?
   - Logical Errors: These errors prevent the program from functioning correctly, and resolving them is crucial for proper execution.
3. Which type of error were you not able to identify using program inspection?
   - Runtime Errors: These could arise due to incorrect recursive calls and lead to stack overflow or unexpected behavior.

4. Is the program inspection technique worth applying?
    - Yes, it is essential for identifying logical errors and ensuring that the algorithm works correctly.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.
    - Two errors were identified:
        - Incorrect handling of parameters in recursive calls.
        - Modifying peg names during recursion, which is logically incorrect.
2. How many breakpoints did you need to fix those errors?
    - Two breakpoints:
        - One at the beginning of the doTowers method to check the values of topN, from, inter, and to.
        - One inside the condition to ensure proper disk movements are printed.

    a. What are the steps you have taken to fix the errors you identified in the code fragment?

    - Fix 1: Changed doTowers(topN++, inter--, from + 1, to + 1) to doTowers(topN - 1, from, to, inter) for correct parameter handling.
    - Fix 2: Ensured that the parameters from, inter, and to are passed unchanged to maintain correct peg references.

3. **Submit your complete executable code:**

```java
// Tower of Hanoi

public class MainClass {

  public static void main(String[] args) {

     int nDisks = 3;

     doTowers(nDisks, 'A', 'B', 'C');

  }

  public static void doTowers(int topN, char from, char inter, char to) {

     if (topN == 1) {

        System.out.println("Disk 1 from " + from + " to " + to);

     } else {

        doTowers(topN - 1, from, to, inter); // Move disks from 'from' to 'inter'

        System.out.println("Disk " + topN + " from " + from + " to " + to);

        doTowers(topN - 1, inter, from, to); // Move disks from 'inter' to 'to'

     }

  }

}
```

Output :

Disk 1 from A to C

Disk 2 from A to B

Disk 1 from C to B

Disk 3 from A to C

Disk 1 from B to A

Disk 2 from B to C

Disk 1 from A to C

10. Armstrong :

Program Inspection :

1. How many errors are there in the program? Mention the errors you have identified.
    - Logical Errors:
        - Error 1: The calculation of remainder is incorrect. It should use num % 10 to get the last digit, not num / 10.
        - Error 2: The calculation of check should be done with the remainder, not the updated num. The code mistakenly uses the wrong order in the while loop.
        - Error 3: The check for Armstrong numbers needs to consider that the number of digits in the input must be taken into account for the power calculation.
2. Which category of program inspection would you find more effective?

    - Logical Errors: Identifying logical errors is crucial as they prevent the program from functioning correctly.
3. Which type of error were you not able to identify using program inspection?
    - Runtime Errors: These could arise from incorrect logic and lead to unexpected results or crashes.
4. Is the program inspection technique worth applying?
    - Yes, it is essential for identifying logical errors and ensuring the algorithm behaves as intended.

Debugging :

1. How many errors are there in the program? Mention the errors you have identified.
    - Three errors were identified:
        - Incorrect calculation of the last digit.
        - Incorrect usage of variables in calculations.
        - Failure to account for the number of digits in the Armstrong number logic.
2. How many breakpoints did you need to fix those errors?
    - Three breakpoints:
        - One before the while loop to inspect the initial value of num.
        - One inside the loop to check the value of remainder.
        - One after the loop to verify the final value of check.

a. What are the steps you have taken to fix the errors you identified in the code fragment?

- Fix 1: Changed remainder = num / 10; to remainder = num % 10; to correctly extract the last digit.
- Fix 2: Updated the calculation of check to check = check + (int)Math.pow(remainder, 3); using the correct remainder.
- Fix 3: Added a way to calculate the number of digits in num and raised the remainder to that power instead of always cubing it.

3. Submit your complete executable code:

```
// Armstrong Number

class Armstrong {

   public static void main(String args[]) {

       int num = Integer.parseInt(args[0]);

       int n = num; // use to check at last time

       int check = 0, remainder, digits = String.valueOf(num).length();

       while (num > 0) {

           remainder = num % 10; // Correctly extract the last digit

           check = check + (int) Math.pow(remainder, digits); // Use the number
of digits for power

           num = num / 10; // Update num to remove the last digit

       }
```

```
        if (check == n)

            System.out.println(n + " is an Armstrong Number");

        else

            System.out.println(n + " is not an Armstrong Number");

    }

}
```

Input :

153

Output :

153 is an Armstrong Number.

Static Analysis Tools

Github link :

https://github.com/martinus/robin-hood-hashing/blob/master/src/include/robin_hood.h

Category A: Data Reference Errors

1. **Uninitialized Variables**: Does a referenced variable have a value that is unset or uninitialized? For each reference to a data item (variable, array element, field in a structure), informally prove that the item has a value at that point.
2. **Array Bounds**: For all array references, is each subscript value within the defined bounds of the corresponding dimension?
3. **Integer Subscripts**: For all array references, does each subscript have an integer value? While this is not necessarily an error in all languages, it is a dangerous practice.
4. **Dangling References**: For all references through pointer or reference variables, is the referenced memory currently allocated? This includes checking for situations where a pointer references a local variable that may no longer exist after the procedure returns.
5. **Alias Name Attributes**: When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names?
6. **Variable Type and Attributes**: Does a variable's value have a type or attribute other than what the compiler expects? This can occur when a program reads a record into memory and

references it with a structure that doesn't match the physical representation.

7. **Addressing Problems**: Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory addressability?
8. **Memory Location Attributes**: If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects?
9. **Consistent Data Structures**: If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?
10. **String Indexing**: When indexing into a string, are the limits of the string accurate to avoid off-by-one errors?
11. **Inheritance Requirements**: For object-oriented languages, are all inheritance requirements met in the implementing class?

This list focuses solely on the **Data Reference Errors** category while maintaining the context of each point for clarity. Let me know if you need further modifications!

4o mini

You said:

pleae give all category of erros give

ChatGPT said:

ChatGPT

Certainly! Here's a consolidated list of all categories of errors along with their specific checks:

Category A: Data Reference Errors

1. **Uninitialized Variables**: Does a referenced variable have a value that is unset or uninitialized?
2. **Array Bounds**: For all array references, is each subscript value within the defined bounds of the corresponding dimension?
3. **Integer Subscripts**: For all array references, does each subscript have an integer value?
4. **Dangling References**: For all references through pointer or reference variables, is the referenced memory currently allocated?
5. **Alias Name Attributes**: When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes?
6. **Variable Type and Attributes**: Does a variable's value have a type or attribute other than what the compiler expects?
7. **Addressing Problems**: Are there any addressing problems if the units of memory allocation are smaller than the units of memory addressability?
8. **Memory Location Attributes**: If pointer or reference variables are used, does the referenced memory location have the expected attributes?
9. **Consistent Data Structures**: If a data structure is referenced in multiple procedures or subroutines, is it defined identically in each procedure?
10. **String Indexing**: When indexing into a string, are the limits of the string accurate?
11. **Inheritance Requirements**: For object-oriented languages, are all inheritance requirements met in the implementing class?

Category B: Data-Declaration Errors

1. **Explicit Declaration**: Have all variables been explicitly declared?

2. **Understanding Defaults**: If not all attributes of a variable are explicitly stated in the declaration, are the defaults well understood?
3. **Proper Initialization**: Where a variable is initialized in a declarative statement, is it properly initialized?
4. **Correct Length and Data Type**: Is each variable assigned the correct length and data type?
5. **Memory Type Consistency**: Is the initialization of a variable consistent with its memory type?
6. **Similar Names Warning**: Are there any variables with similar names (e.g., VOLT and VOLTS)?

Category C: Computation Errors

1. **Inconsistent Data Types**: Are there any computations using variables having inconsistent data types?
2. **Mixed-Mode Computations**: Are there any mixed-mode computations that require careful exploration of conversion rules?
3. **Different Lengths**: Are there any computations using variables having the same data type but different lengths?
4. **Target Variable Size**: Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?
5. **Overflow/Underflow Possibility**: Is an overflow or underflow expression possible during the computation of an expression?
6. **Zero Divisor**: Is it possible for the divisor in a division operation to be zero?
7. **Base-2 Representation Issues**: Are there any inaccuracies due to base-2 representation?
8. **Meaningful Range**: Can the value of a variable go outside the meaningful range?

9. **Order of Evaluation**: For expressions with more than one operator, are the assumptions about the order of evaluation correct?
10. **Integer Arithmetic Errors**: Are there any invalid uses of integer arithmetic, particularly divisions?

Category D: Comparison Errors

1. **Different Data Types**: Are there any comparisons between variables having different data types?
2. **Mixed-Mode Comparisons**: Are there any mixed-mode comparisons or comparisons between variables of different lengths?
3. **Correct Comparison Operators**: Are the comparison operators correct?
4. **Boolean Expression Validity**: Does each Boolean expression state what it is supposed to state?
5. **Boolean Operand Types**: Are the operands of a Boolean operator Boolean?
6. **Floating-Point Comparisons**: Are there any comparisons between fractional or floating-point numbers that could lead to errors?
7. **Order of Evaluation for Boolean Operators**: Are the assumptions about the order of evaluation and precedence of Boolean operators correct?
8. **Compiler Behavior**: Does the way in which the compiler evaluates Boolean expressions affect the program?

Category E: Control-Flow Errors

1. **Multiway Branch Indexing**: If the program contains a multiway branch, can the index variable ever exceed the number of branch possibilities?

2. **Loop Termination**: Will every loop eventually terminate?
3. **Module/Subroutine Termination**: Will the program, module, or subroutine eventually terminate?
4. **Non-Executed Loops**: Is it possible that a loop will never execute due to entry conditions?
5. **Loop Fall-Through Consequences**: For loops controlled by iteration and a Boolean condition, what are the consequences of loop fall-through?
6. **Off-By-One Errors**: Are there any off-by-one errors in loops?
7. **Statement Group Matching**: If the language contains statement groups, is there an explicit while for each group and do the do's correspond to their appropriate groups?
8. **Non-Exhaustive Decisions**: Are there any non-exhaustive decisions that make assumptions about input parameters?

Category F: Interface Errors

1. **Parameter and Argument Matching**: Does the number of parameters received by this module equal the number of arguments sent?
2. **Parameter Attribute Matching**: Do the attributes of each parameter match the attributes of each corresponding argument?
3. **Units System Consistency**: Does the units system of each parameter match the units system of each corresponding argument?
4. **Argument Transmission Matching**: Does the number of arguments transmitted by this module to another module equal the number of parameters expected?
5. **Argument Attribute Matching**: Do the attributes of each argument transmitted match the attributes of the corresponding parameter?

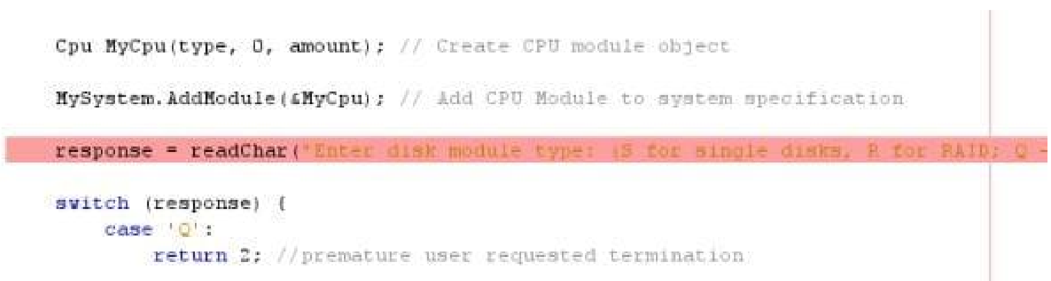
6. **Units System Consistency in Arguments**: Does the units system of each argument transmitted match the units system of the corresponding parameter?
7. **Built-in Function Invocation**: If built-in functions are invoked, are the number, attributes, and order of the arguments correct?
8. **Input-Only Parameters**: Does a subroutine alter a parameter that is intended to be only an input value?
9. **Global Variable Consistency**: If global variables are present, do they have the same definition and attributes in all modules that reference them?

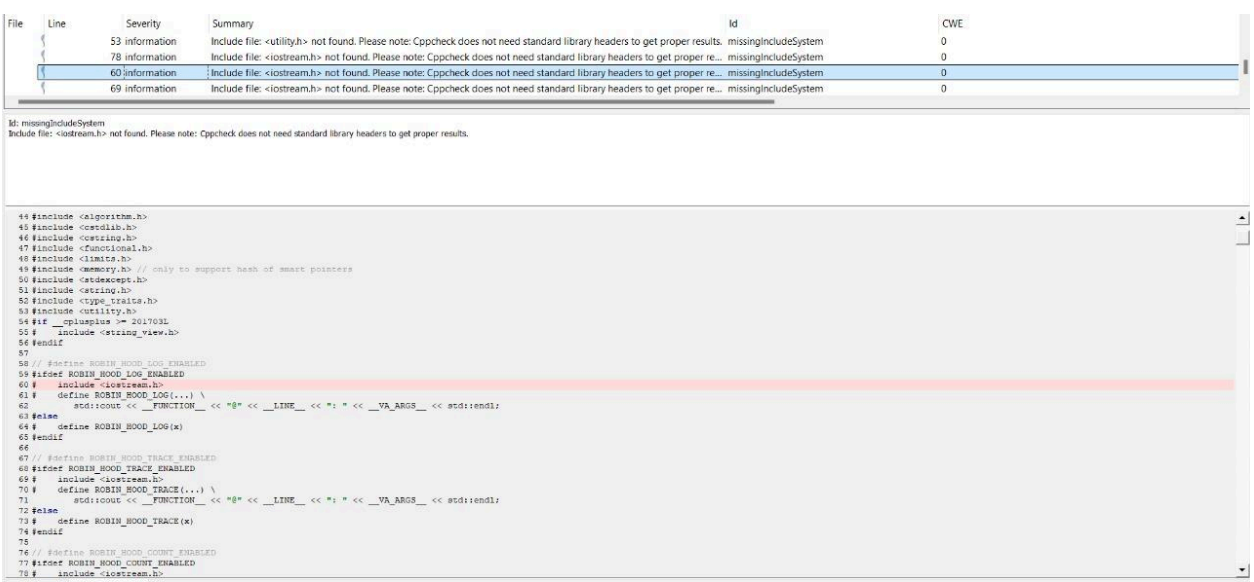
Category G: Input / Output Errors

1. **File Attributes**: If files are explicitly declared, are their attributes correct?
2. **File OPEN Statement**: Are the attributes on the file's OPEN statement correct?
3. **Memory Availability**: Is there sufficient memory available to hold the file your program will read?
4. **File Opening**: Have all files been opened before use?
5. **File Closing**: Have all files been closed after use?
6. **End-of-File Handling**: Are end-of-file conditions detected and handled correctly?
7. **I/O Error Handling**: Are I/O error conditions handled correctly?
8. **Text Errors**: Are there spelling or grammatical errors in any text printed or displayed by the program?

Category H: Other Checks

1. **Cross-Reference Listing**: If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.
2. **Attribute Listing Check**: If the compiler produces an attribute listing, check the attributes of each variable to ensure no unexpected default attributes have been assigned.
3. **Warning Messages**: If the program compiled successfully but produced warning or informational messages, check each one carefully.
4. **Input Validity Check**: Is the program or module sufficiently robust? Does it check its input for validity?
5. **Missing Functions**: Is there a function missing from the program?





```java
public static void main(String[] args) throws java.io.IOException {
  java.io.BufferedReader cin = new java.io.
          BufferedReader(new java.io.InputStreamReader(System.in));
  double s, a, b, S;

  s = 0;
  System.out.print("Enter 1st value: ");
  a = Double.parseDouble(cin.readLine());
  System.out.print("Enter 2nd value: ");
  b = Double.parseDouble(cin.readLine());
  S = a + b;
  System.out.println("The sum is: " + s);
}
```

File	Line	Severity	Summary	Id	CWE
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	53	information	Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	78	information	Include file: <iostream.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <utility.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
67 // #define ROBIN_HOOD_TRACE_ENABLED
68 #ifdef ROBIN_HOOD_TRACE_ENABLED
69 #    include <iostream.h>
70 #    define ROBIN_HOOD_TRACE(...) \
71         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
```

Analysis Log | Warning Details

File	Line	Severity	Summary	Id	CWE
	49	information	Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper re...	missingIncludeSystem	0
	50	information	Include file: <stdexcept.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0
	51	information	Include file: <string.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.	missingIncludeSystem	0
	52	information	Include file: <type_traits.h> not found. Please note: Cppcheck does not need standard library headers to get proper r...	missingIncludeSystem	0

Id: missingIncludeSystem
Include file: <memory.h> not found. Please note: Cppcheck does not need standard library headers to get proper results.

```
33 // OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
34 // SOFTWARE.
35
36 #ifndef ROBIN_HOOD_H_INCLUDED
37 #define ROBIN_HOOD_H_INCLUDED
38
39 // see https://semver.org/
40 #define ROBIN_HOOD_VERSION_MAJOR 3  // for incompatible API changes
41 #define ROBIN_HOOD_VERSION_MINOR 11 // for adding functionality in a backwards-compatible manner
42 #define ROBIN_HOOD_VERSION_PATCH 5  // for backwards-compatible bug fixes
43
44 #include <algorithm.h>
45 #include <cstdlib.h>
46 #include <cstring.h>
47 #include <functional.h>
48 #include <limits.h>
49 #include <memory.h> // only to support hash of smart pointers
50 #include <stdexcept.h>
51 #include <string.h>
52 #include <type_traits.h>
53 #include <utility.h>
54 #if __cplusplus >= 201703L
55 #    include <string_view.h>
56 #endif
57
58 // #define ROBIN_HOOD_LOG_ENABLED
59 #ifdef ROBIN_HOOD_LOG_ENABLED
60 #    include <iostream.h>
61 #    define ROBIN_HOOD_LOG(...) \
62         std::cout << __FUNCTION__ << "@" << __LINE__ << ": " << __VA_ARGS__ << std::endl;
63 #else
64 #    define ROBIN_HOOD_LOG(x)
65 #endif
66
```